FIT2099 - Assignment 3

Recommendations for extending the Engine

Members:

Abhishek Shrestha
Amindu Kaushal

Issue 1

GameMap class in the engine does not allow us to add an exit to another map directly.

We realized that in order to connect the two maps and add exits in between them, we must use the GameMap class from the engine package. However, the provided addExitFromHere() function does not allow adding an exit from location A to location B (where location B is in another map) as it does not take GameMap as a parameter. Since, we were not allowed to change the engine code, when we tried to connect the two maps, we could not simply call a function from GameMap.

Thus, we were forced to create a new class (ParkGameMap) that extends the GameMap class from the engine. Then, we had to create a new addExitfromHereToAnotherMap() function that took in a GameMap as an additional parameter. This unnecessarily increased the number of dependencies made us repeat code, as the two functions are nearly identical. @See ParkGameMap and GameMap

Solution 1

Edit addExitFromHere() method in GameMap so it would take GameMap as an argument. Hence, for future projects, if the designers have to add a new map and connect the two maps, they don't have to inherit the GameMap class for creating a single method. This helps us reduce dependencies and makes code more concise, following the Reduce Dependencies (ReD) and Don't Repeat Yourself (DRY) principles.

Issue 2

Engine does not allow Actors to be added to the map as char in an input List<String> as done with ground (using GroundFactory)

When adding actors to the map, we have to add actors one by one, specifying coordinates and the map. The code to do so is:

```
map.at(x,y).addActor(new Dinosaur());
```

This is very inefficient when there are a lot of actors to add to the map (especially during testing). And to add each actor, we have to write the same code (as shown above) every time. Thus, it does not follow DRY. We can see that adding different types of grounds to the map can be done directly by creating a map as a List<String> with displayChar of each Ground as required in the map, and passing a FancyGroundFactory instance to its constructor. eg.

```
List<String> mapAsString = Arrays.asList(
        "....++",
        "++....",);
GameMap map1 = new GameMap(someFancyGroundFactory, mapAsString);
```

Solution 2

Create an ActorFactory (Similar to FancyGroundFactory) for creating actor instances, given that they have no argument constructors. This option encourages future designers to avoid passing in literals in the constructors, which in turn decreases use of literals. When instantiating the GameMap, we can pass in an instance of this ActorFactory along with fancyGroundFactory. Then, we would be able to add actors as their displayChar (eg. @ for Player) directly to the List<String> map. This would avoid excessive use of literals and reduce code in the Driver significantly.

Issue 3

Inflexible GameMap does not allow creation of different types of Locations

There is a version of Conway's Life in the demo folder, where the implementation of Conway's game of life (changing state of each location between dead and alive) is done in a ConwayLocation class that has been created by inheriting Location. This is a good use of inheritance. However, we realized that there was a ConwayGameMap class (child class of GameMap), simply to override the makeNewLocation(x, y). This is the code for this function in the GameMap class:

```
protected Location makeNewLocation(int x, int y){
        return new Location(this, x, y);
}
```

The makeNewLocation() function in GameMap (as seen above) is not flexible enough to allow designers to return different types of Locations. This forces designers to inherit GameMap and override this function, if they have to create a Location child class.

Solution 3

Allow makeNewLocation() to take an additional argument of type Location. This allows designers to send the an instance of the location (eg. new ConwayLocation()) they want to return directly from this function. We can retrieve the constructor/instance from this Location object as done in the constructor for FancyGroundFactory (by retrieving and returning getInstance(...)). This makes makeNewLocation() much more flexible, and lets designers avoid creation of an additional GameMap class when creating a Location child class. Thus, this reduces dependencies(ReD) and follows the all classes should be responsible for their own properties principle.

Comments on Good design of the engine

From the “Characteristics of Good Software Design” by David P. Voorhees, the design of the engine had the “simplicity” characteristic, since the amount of software elements used in the design of the engine was fairly low.
It also follows the “Security” characteristic, since most of the variables from classes have been declared as private, ensuring the privacy of data.
Another example of protecting privacy and integrity : the getInventory method of the Actor class and the getItem method of the Location class returns an unmodifiable list, instead of returning the list’s reference. This makes the game more secure, by avoiding privacy leaks**.**

Furthermore, the documentation and the design diagrams were clean and easy to understand, which helped us digest the complexity of the system fairly easily. Due to the good documentation, the task of understanding the relationships between different classes was also much easier.